Racket Programming Assignment #3:

Lambda and Basic Lisp

Thomas Moskal

CSC 344 – Professor Graci

Due date: 3/7/22

Learning Abstract:

This assignment is an exercise in the lambda function and properly using lists in racket programming. Lambda functions are a function that does not relate itself to a variable name. This way, you can quickly create functions in a program without tying up variable names. This makes it easier for a programmer who does not want to stop and create a completely new function. If you can learn when to properly use a Lambda expression you can save time and memory. The second task is practice with sets and basic lisp use. A good demonstration to reference if you have syntax questions. In the third task we altered the Sampler function. I really enjoyed getting to choose colors and consider what else a function like Sampler could be used for.

The final task of this assignment was a piece of code that emulates a deck of cards and can compare those cards in many ways. Eventually, the code determines the classification of a pair of cards in two-hand poker. After doing this assignment, I wondered what it would take to expand the function to take in a full hand of cards and a hand for the house. If I could achieve this, I would have a great program to teach poker noobs the value of the cards in their hand/in play. After floating this idea to an older friend who loves poker, I may have to end up completing such a program.

Task 1 – Lambda

1a.

```
> ( ( lambda ( x ) ( list x (+ x 1 ) (+ x 2 ) ) ) 5 )
'(5 6 7)
> ( ( lambda ( x ) ( list x (+ x 1 ) (+ x 2 ) ) ) 0 )
'(0 1 2)
> ( ( lambda ( x ) ( list x (+ x 1 ) (+ x 2 ) ) ) 108 )
'(108 109 110)
```

1b.

```
> ( ( lambda ( x y z ) ( list z y x ) ) 'red 'yellow 'blue )
'(blue yellow red)
> ( ( lambda ( x y z ) ( list z y x ) ) 10 20 30 )
'(30 20 10)
> ( ( lambda ( x y z ) ( list z y x ) ) "Professor Plum" "Colonel Mustard" "Miss Scarlet")
'("Miss Scarlet" "Colonel Mustard" "Professor Plum")
```

1c.

```
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
3
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
3
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
5
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
5
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
5
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
3
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
4
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
5
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
4
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 3 5 )
3
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
15
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
13
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
17
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
15
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
14
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
15
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
14
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
11
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
11
> ( ( lambda ( x y ) (random x (+ y 1 ) ) ) 11 17 )
14
>
```

Task 2 – List Processing Referencers and Constructors

```
>  ( define languages '(racket prolog haskell rust) )
> languages
'(racket prolog haskell rust)
> 'languages
'languages
> ( quote languages )
'languages
> ( car languages )
'racket
> ( cdr languages )
'(prolog haskell rust)
> ( car ( cdr languages ) )
'prolog
> ( cdr ( cdr languages) )
'(haskell rust)
> ( cadr languages )
'prolog
> ( cddr languages )
'(haskell rust)
> ( first languages )
'racket
> ( second languages )
'prolog
> ( third languages )
'haskell
> ( list-ref languages 2 )
'haskell
> ( define numbers '(1 2 3 ) )
> ( define letters '(a b c ) )
> ( cons numbers letters )
'((1 2 3) a b c)
> ( list numbers letters )
'((1 2 3) (a b c))
> ( append numbers letters )
'(1 2 3 a b c)
> ( define animals '(ant bat cat dot eel ) )
> ( car ( cdr ( cdr ( cdr animals ) ) ) )
'dot
> ( cadddr animals )
'dot
> ( list-ref animals 3 )
'dot
> ( define a 'apple )
> ( define b 'peach )
> ( define c 'cherry )
> ( cons a ( cons b ( cons c '() ) ) )
'(apple peach cherry)
> ( list a b c )
'(apple peach cherry)
> ( define x '(one fish) )
> ( define y '(two fish) )
> ( cons ( car x ) (cons (car (cdr x ) ) y ) )
'(one fish two fish)
> ( append x y )
'(one fish two fish)
>
```

Task 3 – Little Color Interpreter

```
> ( sampler )
(?): ( red orange yellow green blue indigo violet )
violet
(?): ( red orange yellow green blue indigo violet )
indigo
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
yellow
(?): ( red orange yellow green blue indigo violet )
orange
(?): ( aet ate eat eta tae tea )
ate
(?): ( aet ate eat eta tae tea )
tea
(?): ( aet ate eat eta tae tea )
aet
(?): ( aet ate eat eta tae tea )
tea
(?): ( aet ate eat eta tae tea )
eta
(?): ( aet ate eat eta tae tea )
tae
(?): ( 0 1 2 3 4 5 6 7 8 9 )
5
(?): ( 0 1 2 3 4 5 6 7 8 9 )
8
(?): ( 0 1 2 3 4 5 6 7 8 9 )
4
(?): ( 0 1 2 3 4 5 6 7 8 9 )
9
(?): ( 0 1 2 3 4 5 6 7 8 9 )
7
(?): ( 0 1 2 3 4 5 6 7 8 9 )
2
(?): . . user break
```

3a.

Sampler Demo ←

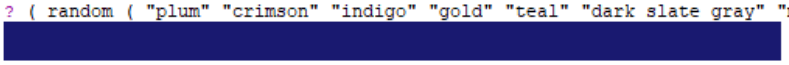
```
( define ( sampler )
   ( display "(?): " )
   ( define the-list ( read ) )
   ( define the-element
      ( list-ref the-list ( random ( length the-list ) ) ) )
   ( display the-element ) ( display "\n" )
   ( sampler )
)
```

←

Sampler Code

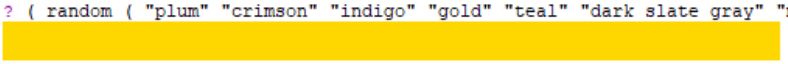
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
> (color-list)
? ( random ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```

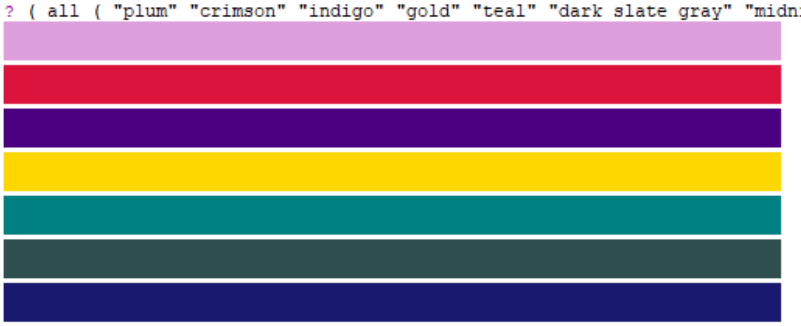


```
? ( random ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```

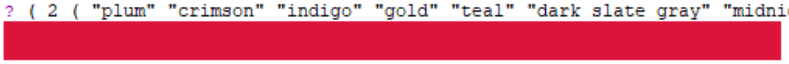
```
? ( random ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```



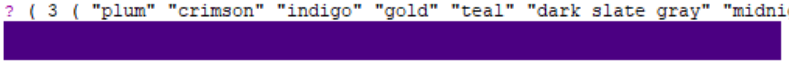
```
? ( all ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```



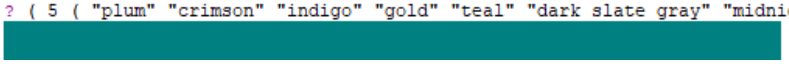
```
? ( 2 ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```



```
? ( 3 ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```



```
? ( 5 ( "plum" "crimson" "indigo" "gold" "teal" "dark slate gray" "midnight blue") )
```



Color-list Demo

```
#lang racket

( require 2htdp/image )

;Makes the color bar
( define ( colored-bar color)
    ( rectangle 500 25 "solid" color)
)

;Recursive helper function for "all" command
(define (action-all list)
  (cond
    ((empty? list) (display "\n") )
    ( else
      (display (colored-bar (car list) ) )
      (display "\n")
      (action-all (cdr list)  )
    )
  )
)

( define ( color-list )
   ( display "? " )
   ( define the-list ( read ) )

     ;Seperate the command from the first element of the original list
   ( define action (car the-list ) )

     ;cadr out the second element of the original list (the list of colors)
   ( define elements (cadr the-list ) )

   ( cond
      ;empty condition : Reread instructions late, didnt want to delete my code that error checks
      ((empty? elements )
        ( display "No elements input" ) )
      ;Random condition : Picks a random color from the list and displays it
      ((equal? action 'random )
         (displayln ( colored-bar ( list-ref elements (random (length elements) ) ) ) ) )
      ;All condition : Displays the entire list in order (using recursion)
      (( equal? action 'all )
        ( action-all elements ) )
      ;Too high of an integer for command :  Reread instructions late, didnt want to delete my code that error checks
      ( (> action (length elements) )
           ( display "Not enough elements in set for this command." )
           ( display " The list only has " ) ( display (length elements) )
           ( display " elements: ")
           ( display elements ) (display "\n") )
      ;Command is an integer : Specifically chooses and displays a color element
      ( (> action 0)
       (displayln ( colored-bar ( list-ref elements (- action  1) ) ) ) )
   )
 (color-list)
)
```

Color-list Code

Task 4 – Two Card Poker

4a. Cards Playing cards Code ->

```
#lang racket

(require racket/trace )

( define ( ranks rank )
   ( list
     ( list rank 'C )
     ( list rank 'D )
     ( list rank 'H )
     ( list rank 'S )
   )
)

( define ( deck )
   ( append
     ( ranks 2 )
     ( ranks 3 )
     ( ranks 4 )
     ( ranks 5 )
     ( ranks 6 )
     ( ranks 7 )
     ( ranks 8 )
     ( ranks 9 )
     ( ranks 'X )
     ( ranks 'J )
     ( ranks 'Q )
     ( ranks 'K )
     ( ranks 'A )
   )
)

( define ( pick-a-card )
   ( define cards ( deck ) )
   ( list-ref cards ( random ( length cards ) ) )
)

( define ( show card )
   ( display ( rank card ) )
   ( display ( suit card ) )
)

( define ( rank card )
   ( car card )
)

( define ( suit card )
   ( cadr card )
)

( define ( red? card )
   ( or
     ( equal? ( suit card ) 'D )
     ( equal? ( suit card ) 'H )
     )
)

( define ( black? card )
   ( not ( red? card ) )
)

( define ( aces? card1 card2 )
   ( and
     ( equal? ( rank card1 ) 'A )
     ( equal? ( rank card2 ) 'A )
   )
)
```

Playing cards Demo

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( define c1 '( 7 c ) )
> ( define c2 '( Q H ) )
> c1
'(7 c)
> c2
'(Q H)
> ( rank c1 )
7
> ( suit c1 )
'c
> ( rank c2)
'Q
> (suit c2)
'H
> (red? c1 )
#f
> (red? c2)
#t
> (black? c1)
#t
> (black? c2)
#f
> (aces? '( A C ) '( A S ) )
#t
> (aces? '( K S ) '( A C ) )
#f
> ( ranks 4 )
'((4 C) (4 D) (4 H) (4 S))
> ( ranks 'K )
'((K C) (K D) (K H) (K S))
> (length (deck) )
52
> ( display (deck ) )
((2 C) (2 D) (2 H) (2 S) (3 C) (3 D) (3 H) (3 S) (4 C) (4 D) (4 H) (4 S) (5 C) (5 D) (5 H) (5 S)
(6 C) (6 D) (6 H) (6 S) (7 C) (7 D) (7 H) (7 S) (8 C) (8 D) (8 H) (8 S) (9 C) (9 D) (9 H) (9 S)
(X C) (X D) (X H) (X S) (J C) (J D) (J H) (J S) (Q C) (Q D) (Q H) (Q S) (K C) (K D) (K H) (K S)
(A C) (A D) (A H) (A S))
> ( pick-a-card )
'(5 D)
> ( pick-a-card )
'(6 H)
> ( pick-a-card )
'(5 C)
> ( pick-a-card )
'(6 H)
> ( pick-a-card )
'(X C)
> ( pick-a-card )
'(K S)
>
```

4b. ur-classifier

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( pick-two-cards )
'((J C) (K H))
> ( pick-two-cards )
'((7 D) (9 D))
> ( pick-two-cards )
'((K D) (5 D))
> ( pick-two-cards )
'((9 S) (6 C))
> ( pick-two-cards )
'((8 D) (3 S))
```

Pick-two Demo

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(5 H) '(K C))
<'K
'K
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(3 S) '(6 D))
<6
6
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(4 H) '(9 D))
<9
9
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(X D) '(J H))
<'J
'J
> ( higher-rank ( pick-a-card ) ( pick-a-card ) )
>(higher-rank '(Q S) '(3 S))
<'Q
'Q
>
```

Higher rank demo

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (classify-two-cards-ur ( pick-two-cards ) )
((6 D) (Q D)): Q high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((X C) (Q S)): Q high
> (classify-two-cards-ur ( pick-two-cards ) )
((9 S) (6 S)): 9 high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((J D) (7 D)): J high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((A S) (2 C)): A high
> (classify-two-cards-ur ( pick-two-cards ) )
((8 H) (4 H)): 8 high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((K C) (3 H)): K high
> (classify-two-cards-ur ( pick-two-cards ) )
((K D) (5 C)): K high
> (classify-two-cards-ur ( pick-two-cards ) )
((9 H) (7 S)): 9 high
> (classify-two-cards-ur ( pick-two-cards ) )
((9 H) (K H)): K high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((9 S) (5 H)): 9 high
> (classify-two-cards-ur ( pick-two-cards ) )
((6 C) (J H)): J high
> (classify-two-cards-ur ( pick-two-cards ) )
((7 C) (X C)): X high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((K H) (A S)): A high straight
> (classify-two-cards-ur ( pick-two-cards ) )
((X H) (8 S)): X high
> (classify-two-cards-ur ( pick-two-cards ) )
((K H) (J S)): K high
> (classify-two-cards-ur ( pick-two-cards ) )
((J C) (9 C)): J high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((Q D) (5 D)): Q high flush
> (classify-two-cards-ur ( pick-two-cards ) )
((8 C) (K H)): K high
> (classify-two-cards-ur  '(( J H )( J D )))
((J H) (J D)): J Pair
>
```

UR classifier demo

```
#lang racket

(require racket/trace )

( define ( ranks rank )
   ( list
     ( list rank 'C )
     ( list rank 'D )
     ( list rank 'H )
     ( list rank 'S )
    )
 )

( define ( deck )
   ( append
     ( ranks 2 )
     ( ranks 3 )
     ( ranks 4 )
     ( ranks 5 )
     ( ranks 6 )
     ( ranks 7 )
     ( ranks 8 )
     ( ranks 9 )
     ( ranks 'X )
     ( ranks 'J )
     ( ranks 'Q )
     ( ranks 'K )
     ( ranks 'A )
   )
)

( define ( rank_conversion n )
   (cond
      ( (equal? n 'X ) 10)
      ( (equal? n 'J ) 11)
      ( (equal? n 'Q ) 12)
      ( (equal? n 'K ) 13)
      ( (equal? n 'A ) 14)
   )
)

( define ( convert_rank card)
   (cond
     (( number? (car card) )
       (car card) )
     (else
        (rank_conversion (car card))
     )
   )
)

( define ( pick-a-card )
   ( define cards ( deck ) )
   ( list-ref cards ( random ( length cards ) ) )
)

( define ( show card )
   ( display ( rank card ) )
   ( display ( suit card ) )
)

( define ( rank card )
   ( car card )
)

( define ( suit card )
   ( cadr card )
)

```

```
( define ( red? card )
   ( or
     ( equal? ( suit card ) 'D )
     ( equal? ( suit card ) 'H )
   )
)

( define ( black? card )
   ( not ( red? card ) )
)

( define ( aces? card1 card2 )
   ( and
     ( equal? ( rank card1 ) 'A )
     ( equal? ( rank card2 ) 'A )
   )
)

( define ( pick-two-cards )
   ( define card1 (pick-a-card) )
   ( define card2 (pick-a-card) )
   ( cond
      ((eq? (rank card1) (rank card2))
               (pick-two-cards))
      (else
              (list card1 card2 ) )
    )
)

( define ( higher-rank card1 card2 )

   ( cond
      ((equal? card1 card2)
       (higher-rank (pick-a-card) (pick-a-card) ) )
      (( > (convert_rank card1) (convert_rank card2) )
         ( car card1 ) )
      (else
         ( car card2 )
      )
    )
)

( define (classify-two-cards-ur ulti-list )
   (display ulti-list )
   (display ": ")
   (define card1 (car ulti-list) )
   (define card2 (cadr ulti-list) )
   (define high ( higher-rank card1 card2 ) )
   ( display high )
   (cond
     ((equal? (cadr card1) (cadr card2) )
      (cond
        ((or (= (- ( convert_rank card1 )  ( convert_rank card2 ) ) 1) (= (- ( convert_rank card2 )  ( convert_rank card1 ) ) 1) )
         ( display " high straight flush" ) )
        (else
         (display " high flush" ) ) ) )
   (else
    (cond
      ((or (= (- ( convert_rank card1 )  ( convert_rank card2 ) ) 1) (= (- ( convert_rank card2 )  ( convert_rank card1 ) ) 1) )
          (display " high straight" ))
      (else
       (cond
         ((equal? (car card1) (car card2) )
          (display " Pair" ) )
         (else
          (display " high" ) ) ) ) ) )
   )
 )
;(classify-two-cards-ur ( pick-two-cards ) )
;(classify-two-cards-ur  '(( J H )( J D )))
;( trace higher-rank )
```

UR classifier code

4c. classifier

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (classify-two-cards ( pick-two-cards ) )
((9 H) (6 S)): Nine high
> (classify-two-cards ( pick-two-cards ) )
((5 C) (7 S)): Seven high
> (classify-two-cards ( pick-two-cards ) )
((4 H) (K D)): King high
> (classify-two-cards ( pick-two-cards ) )
((Q H) (6 S)): Queen high
> (classify-two-cards ( pick-two-cards ) )
((8 S) (3 C)): Eight high
> (classify-two-cards ( pick-two-cards ) )
((6 S) (7 D)): Seven high straight
> (classify-two-cards ( pick-two-cards ) )
((6 C) (3 D)): Six high
> (classify-two-cards ( pick-two-cards ) )
((A D) (3 D)): Ace high flush
> (classify-two-cards ( pick-two-cards ) )
((7 D) (3 D)): Seven high flush
> (classify-two-cards ( pick-two-cards ) )
((Q H) (4 C)): Queen high
> (classify-two-cards ( pick-two-cards ) )
((9 H) (3 H)): Nine high flush
> (classify-two-cards ( pick-two-cards ) )
((3 C) (A C)): Ace high flush
> (classify-two-cards ( pick-two-cards ) )
((3 S) (9 C)): Nine high
> (classify-two-cards ( pick-two-cards ) )
((Q C) (K C)): King high straight flush
> (classify-two-cards ( pick-two-cards ) )
((A H) (8 S)): Ace high
> (classify-two-cards ( pick-two-cards ) )
((A D) (Q H)): Ace high
> (classify-two-cards ( pick-two-cards ) )
((9 H) (8 S)): Nine high straight
> (classify-two-cards ( pick-two-cards ) )
((A C) (2 H)): Ace high
> (classify-two-cards ( pick-two-cards ) )
((6 C) (8 H)): Eight high
> (classify-two-cards ( pick-two-cards ) )
((9 H) (8 C)): Nine high straight
>
```

Classifier Demo

```
#lang racket

(require racket/trace )

( define ( ranks rank )
   ( list
     ( list rank 'C )
     ( list rank 'D )
     ( list rank 'H )
     ( list rank 'S )
    )
 )

( define ( deck )
   ( append
     ( ranks 2 )
     ( ranks 3 )
     ( ranks 4 )
     ( ranks 5 )
     ( ranks 6 )
     ( ranks 7 )
     ( ranks 8 )
     ( ranks 9 )
     ( ranks 'X )
     ( ranks 'J )
     ( ranks 'Q )
     ( ranks 'K )
     ( ranks 'A )
   )
)

( define ( rank_conversion n )
   (cond
      ( (equal? n 'X ) 10)
      ( (equal? n 'J ) 11)
      ( (equal? n 'Q ) 12)
      ( (equal? n 'K ) 13)
      ( (equal? n 'A ) 14)
   )
)

( define ( english_conversion n )
   (cond
     ( (equal? n 2 ) "Two")
     ( (equal? n 3 ) "Three")
     ( (equal? n 4 ) "Four")
     ( (equal? n 5 ) "Five")
     ( (equal? n 6 ) "Six")
     ( (equal? n 7 ) "Seven")
     ( (equal? n 8 ) "Eight")
     ( (equal? n 9 ) "Nine")
      ( (equal? n 'X ) "Ten")
      ( (equal? n 'J ) "Jack")
      ( (equal? n 'Q ) "Queen")
      ( (equal? n 'K ) "King")
      ( (equal? n 'A ) "Ace")
   )
)


( define ( convert_rank card)
   (cond
     (( number? (car card) )
       (car card) )
     (else
        (rank_conversion (car card))
     )
   )
)

( define ( pick-a-card )
   ( define cards ( deck ) )
   ( list-ref cards ( random ( length cards ) ) )
)

( define ( show card )
   ( display ( rank card ) )
   ( display ( suit card ) )
)

```

```
( define ( rank card )
   ( car card )
)

( define ( suit card )
   ( cadr card )
)

( define ( red? card )
   ( or
     ( equal? ( suit card ) 'D )
     ( equal? ( suit card ) 'H )
   )
)

( define ( black? card )
   ( not ( red? card ) )
)

( define ( aces? card1 card2 )
   ( and
     ( equal? ( rank card1 ) 'A )
     ( equal? ( rank card2 ) 'A )
   )
)

( define ( pick-two-cards )
   ( define card1 (pick-a-card) )
   ( define card2 (pick-a-card) )
   ( cond
      ((eq? (rank card1) (rank card2))
               (pick-two-cards))
      (else
              (list card1 card2 ) )
    )
)

( define ( higher-rank card1 card2 )

   ( cond
      ((equal? card1 card2)
       (higher-rank (pick-a-card) (pick-a-card) ) )
      (( > (convert_rank card1) (convert_rank card2) )
         ( car card1 ) )
      (else
         ( car card2 )
      )
    )
)

( define (classify-two-cards ulti-list )
   (display ulti-list )
   (display ": ")
   (define card1 (car ulti-list) )
   (define card2 (cadr ulti-list) )
   (define high ( higher-rank card1 card2 ) )
   (define english_name ( english_conversion high ) )
   (display english_name)
   (cond
     ((equal? (cadr card1) (cadr card2) )
      (cond
        ((or (= (- ( convert_rank card1 )  ( convert_rank card2 ) ) 1) (= (- ( convert_rank card2 )  ( convert_rank card1 ) ) 1) )
         ( display " high straight flush" ) )
        (else
         (display " high flush" ) ) ) )
   (else
    (cond
     ((or (= (- ( convert_rank card1 )  ( convert_rank card2 ) ) 1) (= (- ( convert_rank card2 )  ( convert_rank card1 ) ) 1) )
         (display " high straight" ))
      (else
       (cond
         ((equal? (car card1) (car card2) )
          (display " Pair" ) )
         (else
          (display " high" ) ) ) ) ) )
   )
 )
;(classify-two-cards ( pick-two-cards ) )
;(classify-two-cards '(( A H )( A D )))
;( trace higher-rank )
```

Classifier Code